

# A Framework for Caching in an Object-Oriented System

Michael N. Nelson  
Graham Hamilton  
Yousef A. Khalidi

SMLI TR-93-19

October 1993

## Abstract:

Caching is an important technique for improving performance in distributed systems. However, in general, it has been performed on an ad-hoc basis, with each component of the system having to invent its own caching techniques. In the Spring operating system, we provide a unified caching architecture that can be used to cache a variety of different kinds of remote objects. For any given kind of object, this architecture lets different client processes within a single machine share a single cache for accessing remote objects. This caching is performed by a separate cacher process on the machine local to client processes, the caching is transparent to the clients, and the cached information is kept coherent. This architecture has been used to implement caching for files and for naming contexts.

 *Sun Microsystems*  
*Laboratories, Inc.*

A Sun Microsystems, Inc. Business

M/S 29-01  
2550 Garcia Avenue  
Mountain View, CA 94043

## email addresses:

michael.nelson@eng.sun.com  
graham.hamilton@eng.sun.com  
yousef.khalidi @eng.sun.com

# A Framework for Caching in an Object-Oriented System

*Michael N. Nelson   Graham Hamilton   Yousef A. Khalidi*

Sun Microsystems Laboratories, Inc.  
2550 Garcia Avenue,  
Mountain View, CA 94043 USA

## 1 Introduction

---

Caching of file and naming information has been used in operating systems for many years. It was originally used in timesharing systems to avoid disk operations (e.g., the UNIX<sup>®</sup> operating system [1]) and was later used in distributed systems to reduce network accesses and server loading (e.g., the Andrew [2] and Sprite [3] file systems). Measurements of distributed systems [2, 3] have shown caching to be very effective in improving performance. In fact, in many cases caching can allow access to remote information to be as efficient as access to local information.

This paper describes a general architecture for caching in Spring, a distributed operating system that is structured around the notion of objects. A Spring object is an abstraction that contains state and provides a set of operations to manipulate that state. Spring objects are implemented by *domains* which consist of an *address space* and a collection of threads; a domain in Spring is similar to a process on other systems such as the UNIX operating system. Objects can be freely passed between domains. Object invocations go to the implementor of the object which can be the same domain that is invoking the operation, another domain on the same machine, or a domain on a different machine.

In the Spring operating system, we use caching to improve the performance of operations on remotely implemented objects. Our goal is to make operations on remotely implemented objects as efficient as operations on locally implemented objects. In order to implement caching on Spring, we developed an architecture for caching that has several interesting features. First, the architecture is designed to be

used to implement caching with coherency for many types of objects. This is important because new types of objects can be introduced into the Spring operating system at any time and we want to make the benefits of caching easily available to implementations of these new objects. This is a different approach from that taken in systems such as the UNIX and Sprite operating systems where the number of cacheable objects is fixed and no general caching architecture is available.

The second interesting feature of the Spring caching architecture is that caching is performed by a separate cacher domain. This is a different approach than that used in object oriented systems such as [4, 5, 6, 7] where each domain caches a copy of the objects that it is using. These systems are all designed around a particular object-based programming system that supports distributed programming. Their models assume a single multi-threaded address space where any operation outside the address space is considered to be remote and therefore worthy of caching.

We use a separate domain for caching in the Spring operating system because the Spring operating system uses a different model from that used in single-address-space systems. In the Spring operating system, a machine has many domains, each with its own address space. Since the goal of Spring caching is to avoid crossing machine boundaries whenever possible, we want all domains on a machine to be able to share cached information from remote operations. We also want cached information to persist across the life-span of domains so that cached information will not have to be reacquired every time a

new domain is created that uses a remotely implemented object.

The third interesting feature of the Spring caching architecture is that caching is completely transparent to client applications. If an application program happens to use an object that supports caching, then the object will be automatically cached, without the application needing to be aware of it. Some previous implementations have provided this feature [5, 8] and others have not [4, 6, 7].

The fourth interesting feature of the Spring caching architecture is that it does not rely on Global Unique Identifiers (GUIDs) for object equivalency. The ability to determine whether objects are equivalent is necessary so that the system can decide what objects should share cached information. In systems such as CHORUS<sup>®</sup> [8] where every object has a global unique identifier, the system can determine if two objects are equivalent merely by seeing if the GUIDs of the two objects are equal. In the Spring operating system, we determined that GUIDs have too many limitations and instead chose to develop as part of the architecture a special secure protocol for determining object equivalency.

We have used our caching architecture to implement caching for both file objects and naming context objects (a naming context object contains name-to-object bindings). The file cache and the naming context cache work together to provide the best possible performance. Measurements of file caching show that it is very effective in improving the performance of remote operations.

The rest of this paper is organized as follows: Section 2 provides an overview of the Spring operating system; Section 3 provides an overview of the caching architecture; Section 4 describes the protocol for allowing objects to be cached; Section 5 shows why we chose not to use GUIDs in the Spring operating system and describes the Spring algorithm for determining if an object is already being cached; Section 6 describes the file caching implementation; Section 7 describes the name caching implementation; Section 8 describes how the file and name caching systems can work together to provide the best performance; Section 9 presents related work; Section 10 gives the current implementation status and future work; Section 11 gives some performance measurements; and Section 12 offers some conclusions.

## 2 The Spring Operating System

In this section, we will provide a short overview of the Spring operating system. We will only present those fea-

tures that are necessary to understand the Spring caching architecture.

The description of Spring objects and their operations are specified in an *interface definition language* (IDL). IDL supports both notions of *single* and *multiple* interface inheritance.

Spring objects consist of two parts: the object representation that lives in the domain that is using the object and the state kept by the implementor of the object (we also refer to the implementor of the object as the object manager). The object representation contains at least enough state to allow an invocation on the object to get to the object implementor. Figure 1 shows an example of a Spring object where the client of the object and the server of the object are on different machines.

The Spring kernel supports basic cross domain invocations and threads, low-level machine-dependent handling, as well as basic virtual memory support for memory mapping and physical memory management [9, 10, 11]. A Spring kernel does not know about other Spring kernels—all remote invocations are handled by a *network proxy* server.

A typical Spring node runs several servers besides the kernel. These include a name server; file servers; a linker domain that manages and caches dynamically linked libraries [12]; a network proxy that handles remote invocations; a tty server that provides basic terminal handling as well as frame-buffer and mouse support; and a UNIX server that provides support for running UNIX binaries on the Spring operating system [13].

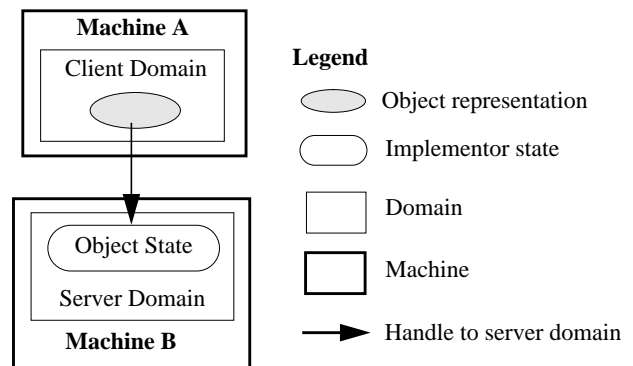


FIGURE 1. Spring object

The client domain has an object that is implemented by a server domain. The client has a representation for the object that allows the invocations on the object to get to the server domain. The server keeps some state for the object.

## 2.1 Spring Security

If the implementor and the client are in different domains, the representation of an object includes an unforgeable *handle* managed by the kernel that identifies the server domain. These unforgeable handles have many of the security properties of capabilities [14]. If a server determines that a client is entitled to specific access to a given piece of state (e.g., a file) it can give the client an unforgeable handle X. Encapsulated in the server side state for handle X will be the granted access rights and possibly the principal name of the client. Whenever a call arrives quoting handle X, the server can permit the given access to the underlying state without further checks.

Authenticated Spring objects can be freely passed between domains. Once an object has been issued that contains a set of encapsulated rights, anyone to whom the object is passed will be allowed the same rights.

## 2.2 Spring Naming

The Spring naming service allows any object to be associated with any name. A name-to-object association is called a *name binding*. Each name binding is stored in a context. A *context* is an object that contains a set of name bindings in which each name is unique. Any object, including a context object, can be bound to several different names in possibly several different contexts at the same time. Objects are retrieved from a context by invoking the *resolve* operation, bound by invoking the *bind* operation, and removed by invoking the *unbind* operation. There are also operations to obtain a list of all bindings in a context, to create contexts, to retrieve multiple objects at once, and to get the attributes of a context.

## 2.3 Object Equivalency

In the Spring operating system, we regard the notion of object equivalency as a type dependent property. Different types may choose to provide different definitions of object equivalency to reflect how their objects are used. In fact a particular type may have multiple definitions of object equivalency. For example, a file object may have two definitions of equivalency. The first definition could be that two file objects are said to be equivalent if they share the same underlying server state; however, the two objects may have different encapsulated access rights. A second definition could be that two file objects are said to be equivalent only if they share the same underlying server state and they have the same encapsulated access rights.

## 3 Caching Architecture Overview

---

In this section, we will provide a brief overview of the caching architecture. We will discuss each component in detail in subsequent sections of the paper.

The domains involved when the caching architecture is used to cache an object are the client domain that is using the object, the server domain that implements the object, and the cacher domain that is caching the object. The caching architecture consists of three components that these domains use together to implement caching:

- A mechanism called the *caching subcontract* that allows client domains to transparently contact cacher domains to cache objects (Section 4)
- A *bind* protocol that the cacher domain and the server domain follow to implement caching (Section 5)
- A set of objects that are implemented by the cacher and server domains (Section 3.1)

Client domains do not have to change to use the caching architecture. Client domains merely link in generic library code that implements the client side of the *caching subcontract* and caching will automatically occur. This library code will work on all object types and requires no effort on the part of implementors of cacheable objects.

The cacher domain and server domain code for caching, on the other hand, must be reimplemented for each object type. This is because, in general, each object type has unique caching needs. For example, file objects provide read and write operations, and naming contexts provide a name resolution operation. The details of caching the results of naming context operations are very different from the details of implementing caching of read and write operations. However, there are some similarities between different caching implementations that could potentially be implemented in a library (see Section 10).

Fortunately, we provide an architecture that when used will make it straightforward to implement and cache cacheable objects. In addition, by looking at examples of existing caching implementations that use the architecture, it is straightforward to produce a caching implementation for a new object type.

### 3.1 Architecture Configuration

Figure 2 shows the basic configuration when there is no caching. In this configuration, there is an object of type *cacheable* that is implemented by the *cacheable object manager (COM)*. The client domain has a copy of this

cacheable object. When the client invokes an operation on the object the invocation goes to the COM.

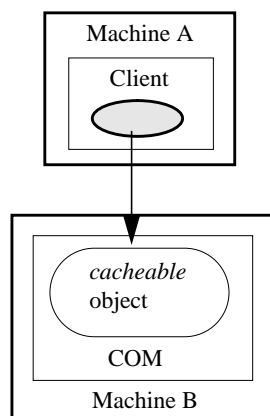


FIGURE 2. No caching

A client domain has a cacheable object that is implemented by the cacheable object manager (COM) domain.

Figure 3 shows the configuration when an object is cached using the caching architecture. The purpose of the cacher domain is to interpose on invocations from a client domain on a cacheable object. The *cacher* object implemented by the cacher domain is used to set up this interposition. When a client domain wants to have a cacheable object cached, it invokes an operation on the cacher object giving the cacher a copy of the cacheable object. The cacher responds with a *cached* object that it implements. This cached object is then used by the client domain for future invocations. When the cacher domain gets an invocation on a cached object, it can either handle the invocation itself or it can forward the request to the COM using the cacheable object or the *provider* object.

The cached information kept by the cacher domain must be kept coherent. This is the purpose of the *provider* and *cache* objects. When the cacher domain needs to get cached information, it invokes an operation on the *provider* object; and when the COM needs to retrieve cached information, it invokes an operation on the *cache* object. Thus the *provider* and *cache* object provide a two-way coherency connection between the cacher domain and the COM. This two-way connection is set up using the *cacher* object and the cacheable object.

## 4 Caching Subcontract

Every Spring object type has an associated subcontract [15]. A subcontract is executable code that is responsible

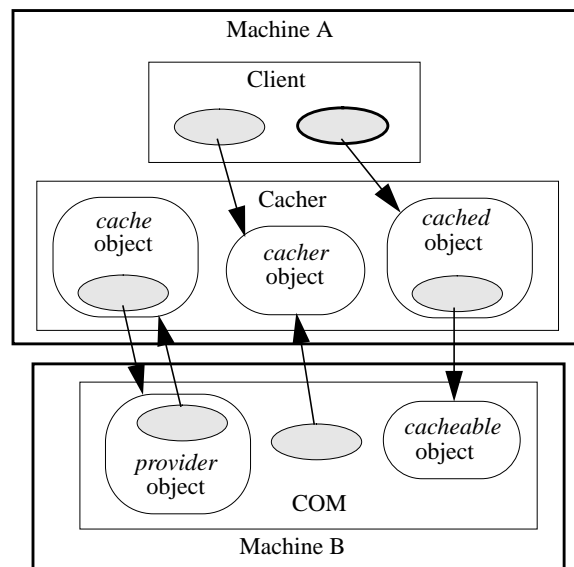


FIGURE 3. Caching with the caching architecture

The client domain has a cached object and a cacher object that are both implemented by a cacher domain. The cacher object was used to get the cached object. The cacher domain has a cacheable object that is implemented by the COM domain. In addition there is a two-way coherency connection between the cacher and the COM consisting of a *provider* and a *cache* object. The COM also has a copy of the cacher object that it uses to setup the coherency connection.

for many things including marshaling, unmarshaling, and invoking operations on the object. Subcontract also defines the representation for each object that appears in a client domain's address space. When an object is invoked, marshaled, or unmarshaled, the subcontract code is executed to perform the operation.

The most widely used Spring subcontract is called *singleton*. The representation of a singleton object consists of a kernel handle that identifies the server domain. When a client invokes an object that uses *singleton*, this handle is used to send the invocation to the server domain.

Cacheable objects use a different subcontract called the *caching* subcontract. The caching subcontract representation contains a handle that identifies the server domain, an object of type *cached* that is implemented by a cacher domain, and the name of the cacher to use (see Figure 4). The cached object and the server handle are used when an invocation occurs. If the cached object is null, then an invocation is done using the server handle. Otherwise, an invocation is done using the cached object. The cached object is null if there is no cacher, if the server is on the same machine as the invoking domain, or if the cacher needs to use a cacheable object.

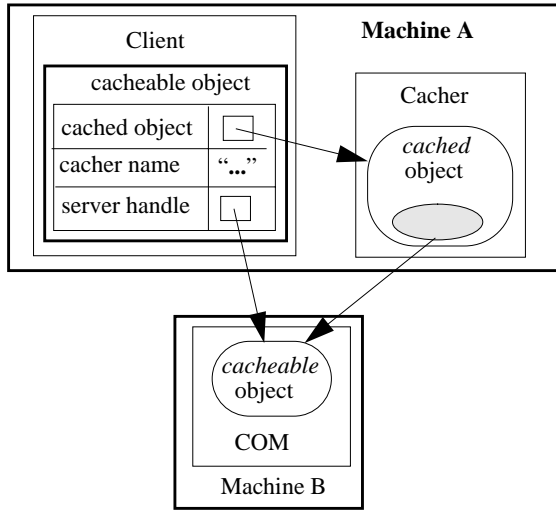


FIGURE 4. State after object is cached

The representation of a cacheable object consists of a cached object that is implemented by a cacher domain, a cacher name that names the cacher to use, and a handle to the server domain.

The cached object is obtained when an object is unmarshaled into a client domain (see Figure 5). The unmarshaling code first resolves the name in the representation to a cacher object. The unmarshaling code then invokes the *get\_cached\_obj* operation on the cacher object passing in a copy of the cacheable object being unmarshaled; an object of type *cached* is returned. When the cacher domain receives a cacheable object, it creates a new *cached* object and returns the object to the client domain. The object returned from the cacher is stored as the *cached* object in the cacheable object's representation.

The name of the cacher object in the representation must be agreed upon by the implementor of a cacheable service and the implementors of cacher objects for the service. The name is placed in the object's representation by the COM when it creates a cacheable object, and the cacher domain for a cacheable object binds a cacher object under the name in the local machine's name space.

#### 4.1 Efficiency Considerations

The mechanism that we have described requires three object invocations on each unmarshal of a cacheable object: one to find the cacher object to use, one to contact the cacher, and one to determine whether the cacheable object is equivalent to some other already cached object (see Section 5.2). However, there are three cases that we

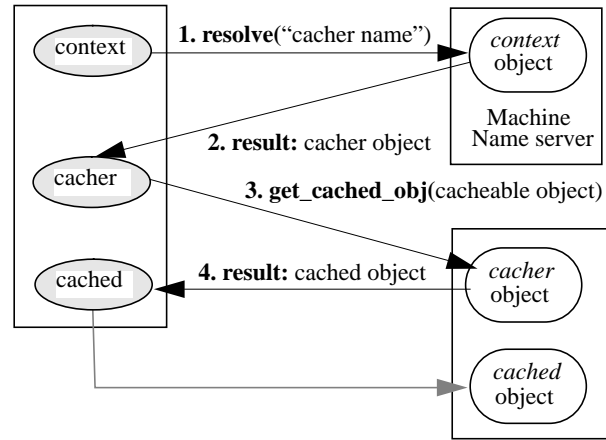


FIGURE 5. Steps to caching an object

have optimized in order to eliminate some or all of these calls.

The first case we optimized is when the cacheable object's server is on the same machine as the domain doing the unmarshal. In this case, the caching subcontract will not even bother to lookup the cacher object. Thus, in this case, no object invocations are required on an unmarshal of a cacheable object.

A second case we optimized occurs when an object that is already cached is being passed between two domains on the same machine. Subcontract does not know when it is asked to marshal an object for a cross-domain call if the destination is on the local machine. As a result, the caching subcontract always marshals both the server handle and the cached object. When a cacher object is invoked with an object to cache it checks if the cached object in the representation is already implemented by the cacher. If so, then the cacher just returns the cached object as the result from the *get\_cached\_obj* call. This optimization eliminates the object invocation used by the cacher domain to determine if a cacheable object is equivalent to some other already cached object.

The third case we optimized occurs when the cacher object has already been acquired from the name space because of a previous unmarshal. This is done by having each domain cache a copy of names and their associated cacher objects from previous unmarshals. In subsequent unmarshals the caching subcontract will try to use a cached copy of the cacher object instead of resolving the cacher name thus eliminating one object invocation.

## 5 The Bind Protocol

When a cacher receives an object to cache, it needs to answer three questions:

- Is this object equivalent to an object that is already being cached? If so, the new object can share the cached state of the equivalent object.
- What are the encapsulated access rights in the cacheable object? These are needed for access control to the shared cached state.
- What provider object should be used to get cacheable data and keep the data coherent?

The first question, “Are two objects equivalent?” can be answered in most systems by using global unique identifiers (GUIDs). In this section, we will first discuss the problems with GUIDs and why we chose not to rely on them, and then we will describe the solution in the Spring operating system that simultaneously answers all three questions.

### 5.1 Problems with Global Unique Identifiers

Some existing systems use GUIDs for objects (e.g., CHORUS [8] and Amoeba [16]) that can be used to determine object equivalency. If each object has a GUID, then two objects can be considered equivalent if the GUIDs of the two objects are equal. However, we believe that this approach to object equivalency is flawed. There are many cases where the flaws in GUIDs show up, but in this section, we will concentrate on four examples.

#### 5.1.1 Interposition

The idea with interposition is that the object used by the client is some intermediate object that intercepts some operations defined in the interface and passes the rest of the operations on to the “real” object. Depending on the situation, the client or the implementation or both may be unaware that interposition has happened. For example, if the interposed object is logging operations for debugging purposes, it is important that neither the client nor the implementation behave differently when being debugged.

If every object has a GUID, then an object  $O$  that is being interposed on by an object  $I$ , will appear to have a different GUID than the object  $O$  itself. Thus, for purposes of object equivalency, these two objects will be considered different. This is not the behavior that we want when a program is being debugged.

#### 5.1.2 Replication

Replication is a common technique for performance, reliability, availability, and other purposes in distributed sys-

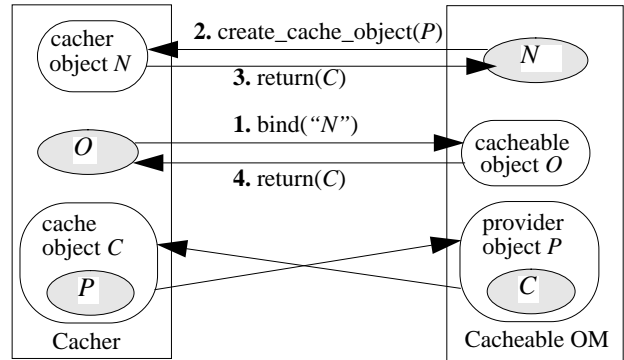


FIGURE 6. Bind protocol

tems. The goal of replication is to allow multiple implementations to provide equivalent services. For example, a client may have two file objects for the same file data yet each is implemented on different servers. For purposes of object equivalency, these objects should be equivalent, but if each object is given a GUID, then they won't be.

#### 5.1.3 Multiple Identities

Some objects play multiple roles and thus have multiple identities. In this case, it is not possible to define one notion of equivalency for these objects. Two objects for some purposes may need to be considered equivalent where for other purposes they should be considered distinct. For example, if two file objects refer to the same on-disk file but have different encapsulated rights, should they be considered equivalent or not? To a cacher they are equivalent since they share the same underlying state, but to a client they may not be equivalent because one object has fewer rights than the other.

#### 5.1.4 Fine Grained Objects

The Spring operating system allows fine grained objects. For example, a complex number could be represented as an object. For fine grained objects we do not want the overhead of creating and storing a GUID.

## 5.2 Bind Protocol

For the reasons given above, the Spring operating system does not use GUIDs to determine object equivalency. Instead, we must involve the object managers for the two objects in question to determine if they are equivalent. We have implemented a special protocol called the *bind* protocol that in a secure manner simultaneously determines if two objects are equivalent for caching purposes, retrieves the encapsulated access rights of the object being cached,

and sets up the provider-cache object connection. Thus, a cacher is able to determine the information necessary for caching using a single secure protocol. This protocol was originally developed for the virtual memory and file system in the Spring operating system and is described in more detail in [11].

Every cacheable subtype defines a *bind* operation. Whenever a cacher is given an object to cache, the cacher invokes the bind operation on the cacheable object. The bind protocol is shown in Figure 6 and involves the following steps:

- A cacher domain issues a bind call on a cacheable object  $O$  passing in a name  $N$  that can be resolved to a *cache* object implemented by the cacher.
- The COM checks if a cacher named  $N$  is already caching an object equivalent to  $O$ <sup>1</sup>.
- If the cacher is already caching an object equivalent to  $O$ , then the COM returns the appropriate cache object.
- Otherwise, the COM resolves  $N$  to a cacher object.
- The COM then invokes the *create\_cache\_object* operation on the cacher object, passing a provider object implemented by the COM as a parameter. The cacher responds by returning a new cache object  $C$ .
- The COM domain remembers the association between  $C$  and  $N$  so that future binds for objects equivalent to  $O$  from the cacher named  $N$  will use the same cache.
- Once the COM has a cache object  $C$  for the given cacher, it returns  $C$  as the result of the bind call.

When a cacher receives a cache object as the result of a bind operation, the cacher can extract its own internal cache information from the cache object. This information contains a pointer to cached information including a provider object to use to acquire additional cached information. If a cacher is already caching an object that it was asked to cache, the bind operation will return a pointer to a cache that can be shared between the two equivalent objects. This has solved two of our problems: we now can determine if we are already caching an object, and we have a provider object to use to get cacheable information.

The other thing that a cacher has to do is determine the encapsulated rights of a cacheable object. Unfortunately, for security reasons, the rights cannot be returned from the bind operation as a forgeable bit string. In fact, returning a cache object as the result of the bind is also not secure.

1. The definition of object equivalency is up to the COM to decide. The definitions of equivalency for the file and name cachers are given in Sections 6.4 and 7.4 respectively.

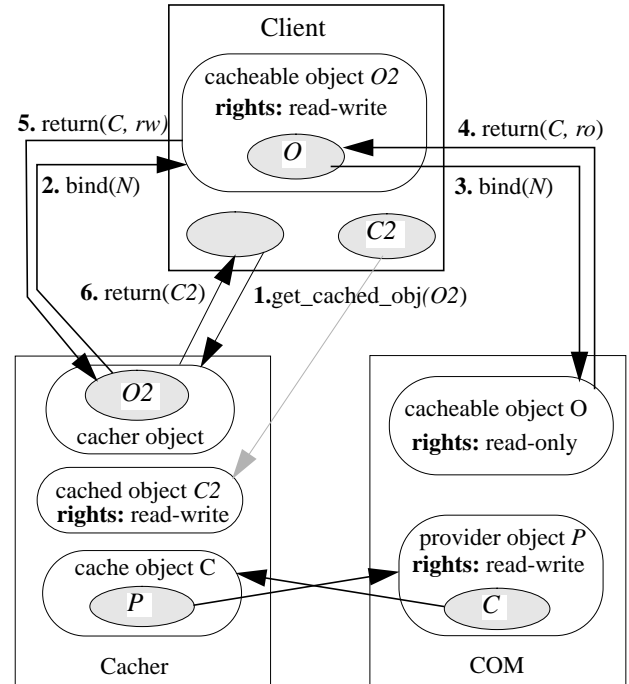


FIGURE 7. Interposer error

Figure 7 shows an example of the security problems that can occur with an interposer with the version of the bind operation that we have described so far. For this example, a client domain is assumed to have acquired a read-only version of a cacheable object  $O$  from the COM. In addition, the cacher domain and the COM are assumed to already have a provider-cache object connection where the provider object grants read-write access; thus the cacher domain is allowed to modify the contents of cacheable object  $O$  via the provider object. The security violation occurs after the following steps:

1. A client invokes the *get\_cached\_obj* operation on a cacher object, passing in an object  $O2$  implemented by the client.
2. The cacher invokes the *bind* operation on the cacheable object passing in the cacher's name  $N$ .
3. The client gets the bind request and forwards it on object  $O$  to a COM domain.
4. The COM domain determines that cacher  $N$  already has this object cached read-write and object  $O$  is only allowed read access. The COM returns  $N$ 's cache object  $C$  and indicates  $O$  has read-only access rights.

5. The client takes the result of the bind, keeps a copy of the cache object, and returns the result to the cacher domain changing the access rights to read-write.
6. The cacher domain returns to the client a cached object *C2* that encapsulates read-write access.

After the cacher returns the cached object to the client domain, the client has erroneously been granted the ability to modify cacheable object *O*. The client can modify the object by invoking on either cached object *C2* or on cache object *C*.

We solve the security problem with the bind operation by returning a *cache-rights* object instead of a cache object as the result of a bind operation. A cache-rights object is implemented by a cacher and encapsulates a set of rights to an associated cache. The cacher can use this cache rights object to find the associated cache and discover what rights the cacheable object is allowed; the cache rights object is an unforgeable Spring object and is useless to any domain besides the cacher. Thus, in addition to the cache object, the *create\_cache\_object* call returns an array of <cache rights object, encapsulated access> pairs. The COM domain responds to bind requests by returning a cache rights object that encapsulates the appropriate access rights. The cacher can use the returned cache rights object to discover the allowed rights and associated cache. See Figure 8 for the state after a successful bind has occurred.

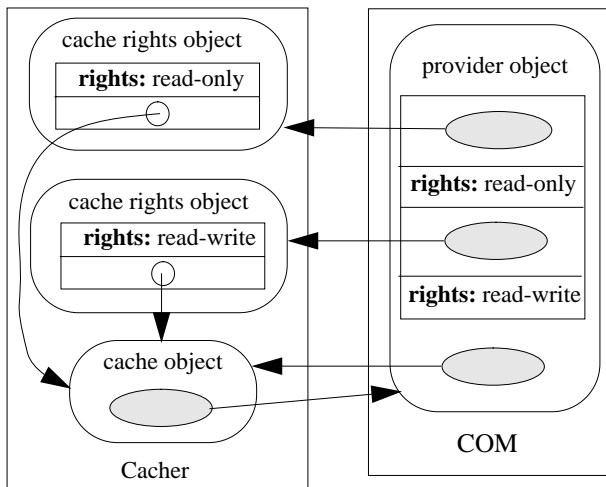


FIGURE 8. State after a successful bind

The COM has two cache-rights objects that encapsulate read-only and read-write access and a cache object all implemented by the cacher. The state for each cache-rights object at the cacher contains the access rights and pointers to shared internal cache state. This state includes a provider object to use.

The list of cache rights objects returned from the *create\_cache\_object* call need not contain objects that encapsulate all possible access rights. A cache rights object supports the *create\_restricted\_sibling* operation that takes a desired set of rights and returns a new cache rights object that encapsulates the given rights; the desired rights must be lower than the actual encapsulated rights in the invoked cache rights object. Thus, as long as the cacher returns at least one cache rights object that encapsulates the highest possible access rights, the COM can create a cache rights object with any needed set of encapsulated rights.

### 5.2.1 Cache and Provider Object Interaction

The cacher issues requests for cacheable information to provider objects. These requests are done on a provider object instead of the associated cacheable object so that the COM can keep track of which cacher is caching information. The COM can use its knowledge of who is caching its information to keep the information coherent by invoking operations on the cache object to retrieve or invalidate cached information.

Each cacheable object type may define its own provider and cache object types that have operations appropriate for the type of object. For example, the provider and cache objects associated with a cacheable file object have operations to retrieve data and attributes and to keep these data and attributes coherent [17]. On the other hand, the provider and cache objects associated with a cacheable naming context object have operations to look up names and keep the cached name-to-object bindings coherent.

## 6 File Caching

One of the cachers that we have implemented using the caching architecture is the file cacher. Each machine that desires to cache files has a Caching File System (CFS) domain running on it. In this section, we will discuss the CFS. The details of file caching are given in [17].

### 6.1 File Types

The file objects that are cached by the CFS are actually of type *cacheable\_file*. The *cacheable\_file* type is a subtype of the file and cacheable types (see Figure 9) and it uses the caching subcontract. The type of object implemented by the CFS is a subtype of *cacheable\_file* called *cached\_file*. Objects of type *cached\_file* use the singleton subcontract.

## 6.2 The CFS Cacher Object

The CFS implements a cacher object that it exports in the Spring naming service. When the CFS receives a *get\_cached\_obj* call, it first determines if it implements the cached object in the representation. If so, it just returns the cached object. Otherwise the CFS follows the previously described bind protocol to find a cache for the cacheable object and discover the cacheable object's encapsulated access rights. Once the bind is done, the CFS constructs an object of type *cached\_file* that it implements and returns it to the client. This *cached\_file* object will encapsulate the proper access rights.

File objects inherit from the Spring *memory object* interface which defines a bind operation that is used by the Spring virtual memory system. The CFS also uses this same bind operation to implement the bind protocol described in Section 5.

## 6.3 FS Cache and Provider Objects

The file system defines its own cache and provider objects which are exchanged as the result of a bind operation. The cache object is of type *fs\_cache* and defines operations to keep file data and attributes coherent; this object is implemented by the CFS. The provider object is of type *fs\_pager*<sup>2</sup> and provides operations to allow the CFS and the associated virtual memory system to get cached attributes and data from the file.

## 6.4 Object Equivalency

Two *cacheable\_file* objects are defined to be equivalent if they share the same server state. However, the two objects may have different encapsulated access rights. For example, two *cacheable\_file* objects that reference the same file on disk are considered to be equivalent, even if one cache-

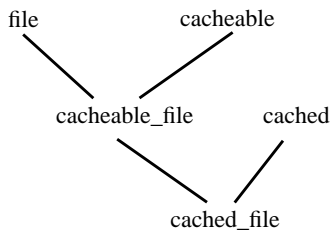


FIGURE 9. File type hierarchy

2. The Spring file system and the VM system refer to a provider object implemented by the file system as a *pager* object.

*able\_file* object grants read-write access and the other *cacheable\_file* object grants read-only access.

## 6.5 Caching

Once the CFS has returned a *cached\_file* object to a client, all future client invocations on the *cacheable\_file* object will be sent to the CFS. The CFS caches file data and file attributes and attempts to respond to all requests using its cache. The CFS actually uses the virtual memory system to cache file data; details are given in [17].

## 7 Name Caching

The other cacher that we have implemented using the caching architecture is the naming context cacher. Each machine that desires to cache naming operations has a Caching Name Server (CNS) domain running on the machine. In this section, we will discuss the name cacher.

### 7.1 Context Types

The context objects that are cached by the CNS are actually of type *cacheable\_context*. The *cacheable\_context* type is a subtype of the context and cacheable types (see Figure 10) and uses the caching subcontract. The *cacheable\_context* type defines a bind operation that can be used by the CNS.

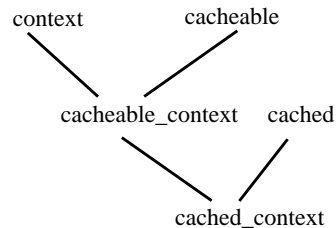


FIGURE 10. Context type hierarchy

The type of object implemented by the CNS is a subtype of *cacheable\_context* called *cached\_context*. Objects of type *cached\_context* use the singleton subcontract.

### 7.2 The CNS Cacher Object

When the CNS receives a *get\_cached\_obj* call, it first determines if it implements the cached object in the representation. If so, it just returns the cached object. Otherwise, the CNS follows the previously described bind

protocol to find a cache for the cacheable object and discover the cacheable object's encapsulated access rights. Once this is done, the CNS constructs an object of type `cached_context` that it implements and returns it to the client. This object will encapsulate the proper access rights.

### 7.3 Name Server Cache and Provide Objects

Name servers (object managers that implement contexts) define their own cache and provider objects that are exchanged as the result of a bind operation. The cache object is of type `ns_cache` and defines operations to keep cached name to object bindings coherent; this object is implemented by the CNS. The provider object is of type `ns_provider` and provides operations to allow the CNS to cache the results of name resolutions.

### 7.4 Object Equivalency

Two `cacheable_context` objects are defined to be equivalent if they share the same server state and the same encapsulated principal. However, the two objects may have different encapsulated access rights.

The encapsulated principal is important for `cacheable_context` equivalency because the encapsulated principal of the object resulting from a resolve operation depends on the encapsulated principal of the context used to perform the resolve. Since the name cacher caches the results of resolve operations, two `cacheable_contexts` cannot share the same cache state unless the two `cacheable_contexts` share the same encapsulated principal.

### 7.5 Caching

Once the CNS has returned the context object to the client, all future client invocations on the cacheable context object will be sent to the CNS. The CNS caches name to object bindings and attempts to respond to all requests from its cache.

## 8 File and Name Cache Interaction

The implementations of caching that we described in sections 6 and 7 require one network access on each cache hit. We would like to eliminate this bind overhead in the normal case. Fortunately, name caching can assist in reducing the bind overhead.

We expect that the Spring naming service will be a very common way for domains to export objects to other

domains. Thus, we expect that many and possibly most objects in the Spring operating system will be acquired from the Spring naming service. The name cacher will cache frequently referenced name to object bindings. In order to avoid paying the bind overhead each time that an object is retrieved from the name cache, the name cacher contacts the local cacher domain when cacheable objects enter the name cache (see [18] for details). Thus, the bind protocol cost is only paid once when a cacheable object first enters the name cache.

## 9 Related work

Caching of objects has been used for many years in many systems. However, the only system that has a mechanism similar to Spring's caching architecture is CHORUS[8]. Like the Spring caching framework, CHORUS also uses a separate domain per-machine to provide file caching. The CHORUS system dynamically binds to a cacher through the use of a *coherent* capability. When an object is created, it contains the known port of the local cache manager. All invocations on the object will be indirected through the local cache manager. Each object is identified by a GUID that can be used by the cacher to determine object equivalency.

Although the CHORUS mechanism is similar to the Spring mechanism, the Spring mechanism has several advantages:

- It provides a general architecture that can be applied to many types of objects; it is unclear from [8] if CHORUS provides a general architecture or an architecture that is specific to file caching.
- It does not require a GUID per cacheable object.
- The cacher is identified by a name instead of a specific port number.

## 10 Status and Future Work

The Spring operating system is currently running native on SPARCstation™ 2 and SPARCstation 10 machines. We have implemented file and name caching using our caching architecture on the Spring operating system. File caching has been in use for over a year and we are currently testing a new version of our name caching implementation. The old version of name caching used all of the parts of the caching architecture except the bind protocol. Our new version has been modified to use the bind protocol instead of the less efficient mechanism that was used in the old version.

One area of future work is in using our caching architecture to implement caching for other object types. Another area of future work is in trying to develop library code that can be shared by the different caching implementations. Since all caching implementations are built with a similar structure by using the caching architecture, we should be able to write a set of libraries that will ease the burden on cacheable object implementors.

## 11 Performance

The goal of the Spring caching architecture is to make caching available to many types of objects so that performance can be improved. The actual effectiveness and performance of caching will depend on the particular type of object that is being cached and on the particular implementation of caching for the object. However, in order to give an indication of the effectiveness of caching, in this section we show some simple measurements of the performance of file caching on the Spring framework.

Table 1 shows the performance of some simple operations on file objects. The column labeled *Without Caching* contains measurements of operations on a file object where the operations are not cached by a local cacher domain; thus, the measurements show the cost of going to a server on a remote machine. The column labeled *With Caching* contains measurements of operations on a file object where all operations are handled by a local cacher domain; thus no remote operations are required. The measurements were taken on two SPARCstation 10/42 machines running Spring native on the hardware.

Operation	Without Caching	With Caching
read 4K	6.6 ms	0.17 ms
write 4K	6.8 ms	0.16 ms
get attributes	2.5 ms	0.11 ms

TABLE 1. File system performance

These measurements show the potential effectiveness of caching. Reading and writing file data is almost 40 times faster with caching than without caching.

## 12 Conclusions

The Spring operating system provides a general caching architecture that can be used to implement caching for all types of objects where caching is appropriate. The caching

architecture would have been very difficult to develop in a non-object-oriented system because the architecture relies on both data encapsulation and object inheritance to achieve its effects. We have used this architecture to implement both file and name caching. We hope to use the architecture on other types of objects as well when we determine that caching would be helpful.

Initial measurements of caching using the caching architecture indicate, as expected, that caching is effective in reducing network overhead. However, the measurements of file caching performance that we have given are only an indication of the effectiveness of caching. We need to perform much more extensive performance evaluation to determine the actual effectiveness of caching on overall system performance. However, we expect, based on the past successes of other systems that have employed caching, that caching will be very effective.

## 13 References

- [1] Thompson, K. "UNIX Time-Sharing System: UNIX Implementation." *Bell System Technical Journal* 57, no. 6 (July-August 1978): 1931-1946.
- [2] Howard, J.H., et al. "Scale and Performance in a Distributed File System." *ACM Transactions on Computer Systems* 6, no. 1 (February 1988): 51-81.
- [3] Nelson, M. N., B. B. Welch, and J. K. Ousterhout. "Caching in the Sprite Network File System," *ACM Transactions on Computer Systems* 6, no. 1 (February 1988): 134-154.
- [4] Jul, E., H. Levy, N. Hutchinson, and A. Black. "Fine-Grained Mobility in the Emerald System." *ACM Transactions on Computer Systems* 6, no. 1 (February 1988): 109-133.
- [5] Bal, H. and A. Tanenbaum. "Distributed Programming with Shared Data," *IEEE Conference on Computer Languages* (October 1988): 82-91.
- [6] Chase, J., et al. "The Amber System: Parallel Programming on a Network of Multiprocessors." *Proceedings of the Twelfth ACM Symposium on Operating System Principles* (December 1989): 147-158.
- [7] Feeley, M. and H. Levy. "Distributed Shared Memory with Versioned Objects." *Proceedings of the 1992 OOPSLA Conference* (October 1992): 247-262.
- [8] Abrosimov, V., F. Armand, and M.I. Ortega. "A Distributed Consistency Server for the CHORUS System," *Proceedings of Symposium on Experiences with Distributed and Multiprocessor Systems* (March 1992): 129-148.
- [9] Hamilton, K. G. and P. Kougiouris. "The Spring Nucleus: A Microkernel for Objects," *Proceedings of the 1993 Summer USENIX Conference* (June 1993): 147-160.

- [10] Khalidi, Y. A. and M. N. Nelson. "The Spring Virtual Memory Architecture," *Sun Microsystems Laboratories Technical Report SMLI TR 93-09* (February 1993).
- [11] Khalidi, Y. A. and M. N. Nelson. "A Flexible External Paging Interface." *Proceedings of the 2nd Microkernels & Other Kernel Architectures Symposium* (September 1993).
- [12] Nelson, M. N. and G. Hamilton. "High Performance Dynamic Linking Through Caching." *Proceedings of the 1993 Summer USENIX Conference* (June 1993).
- [13] Khalidi, Y. A. and M. N. Nelson. "An Implementation of UNIX on an Object-Oriented Operating System." *Proceedings of the 1993 Winter USENIX Conference* (January 1993): 469-480.
- [14] Levy, H. M. *Capability - Based Computer Systems*. Digital Press, 1984.
- [15] Hamilton, G., M. L. Powell. and J. G. Mitchell. "Subcontract: A Flexible Base for Distributed Programming." *Proceedings of Fourteenth ACM Symposium on Operating System Principles* (December 1993).
- [16] Mullender, S. J., et al. "Amoeba - A Distributed Operating System for the 1990's." *IEEE Computer* (May 1990): 45-54.
- [17] Nelson, M. N., Y. A. Khalidi, and P. W. Madany. "Experience Building a File System on a Highly Modular Operating System." *Proceedings of Symposium on Experiences with Distributed and Multiprocessor Systems* (September 1993).
- [18] Nelson, M. N., Y. A. Khalidi, and P. W. Madany. "The Spring File System." *Sun Microsystems Laboratories Technical Report SMLI TR 93-10* (February 1993).

© Copyright 1993 Sun Microsystems, Inc. The SMLI Technical Report Series is published by Sun Microsystems Laboratories, Inc.  
Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

#### TRADEMARKS

Sun, Sun Microsystems, and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc. All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCworks, and SPARCcompiler are licensed exclusively to Sun Microsystems, Inc. CHORUS is a registered trademark of Chorus Systems. All other product names mentioned herein are the trademarks of their respective owners